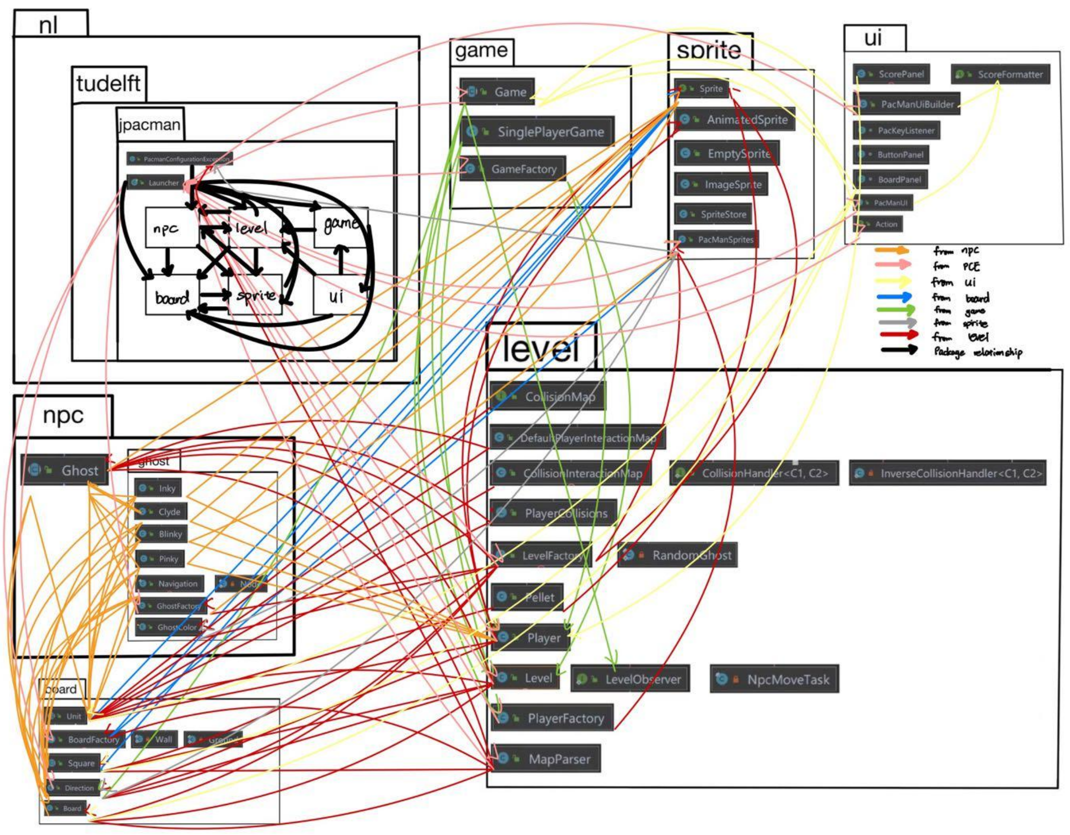
EX1 . question 1

1. Diagram



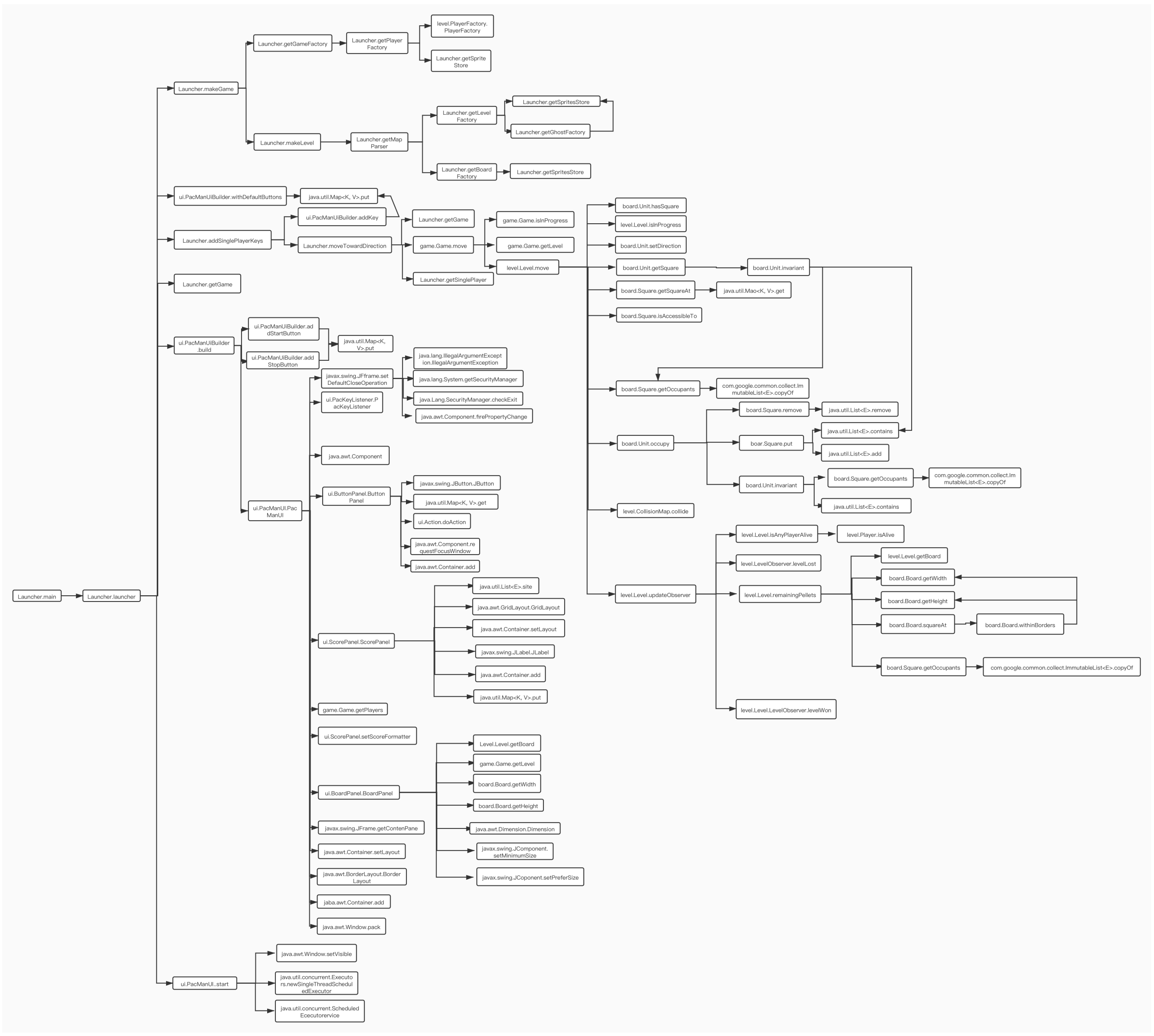
(i) We plan to explain the whole program in 7 level. It is obvious that nl package includes the tudelft package. And tudelft package includes the jpacman package. Meanwhile, we want to talk about the 6 packages and 2classes belong to jpacman package mainly. So the 7 level contains launcher and pacmanconfigurationexception classes (level 1), game package (level 2), sprite package (level 3), ui package (level 4), npc package (level 5), board package (level 6), level package (level7).

(ii) The reason why we choose this kind of classification is that it is easy for us to know the inclusion and hierarchy relationship of packages. So we just talk about the innermost layer, which we mentioned it above. And we already know these 7 levels are included in the jpacman, so we could use the java code to explore the relationship of different packages and classes. What we want to know about the relationships of this java program is described by the diagram we drew. Level and sprite packages influence PacmanConfiguration Exception class (Hereinafter simplified as PCE). Launcher level and sprite packages influence npc package. Launcher, npc, game and ui packages influence level package. Launcher and ui package influence game package. Launcher, npc, level, game, sprite and ui packages influence board package. Launcher, npc, level and board packages influence sprite packages. Launcher influences ui package. And we can get the influence relationship of each class according to the figure above. Arrows of different colors represent the relationship from different levels. We can see the relationship of classes through the figure.

(iii) Different levels may influence each other or unilaterally. Class extends or implements relationships within a package, or they may have relationships across packages. And we can know that launcher class is the start class. We can run the program from this class.

The answer for Question 2 is on the next page

Ex1 -Question2


Launcher.main
Launcher.launcher
Launcher.makeGame
Launcher.getGameFactory
Launcher.getPlayer Factory
level.PlayerFactory. PlayerFactory
Launcher.getSprite Store
Launcher.makeLevel
Launcher.getMap Parser
Launcher.getLevel Factory
Launcher.getSpritesStore
Launcher.getGhostFactory
Launcher.getBoard Factory
Launcher.getSpritesStore
ui.PacManUiBuilder.withDefaultButtons
java.util.Map<K, V>.put
Launcher.addSinglePlayerKeys
ui.PacManUiBuilder.addKey
Launcher.moveTowardDirection
Launcher.getGame
game.Game.move
Launcher.getSinglePlayer
game.Game.isInProgress
game.Game.getLevel
level.Level.move
board.Unit.hasSquare
level.Level.isInProgress
board.Unit.setDirection
board.Unit.getSquare
board.Unit.invariant
board.Square.getSquareAt
java.util.Mao<K, V>.get
board.Square.isAccessibleTo
board.Square.getOccupants
com.google.common.collect.Im mutableList<E>.copyOf
board.Unit.occupy
board.Square.remove
java.util.List<E>.remove
boar.Square.put
java.util.List<E>.contains
java.util.List<E>.add
board.Unit.invariant
board.Square.getOccupants
com.google.common.collect.Im mutableList<E>.copyOf
java.util.List<E>.contains
level.CollisionMap.collide
level.Level.updateObserver
level.Level.isAnyPlayerAlive
level.Player.isAlive
level.LevelObserver.levelLost
level.Level.remainingPellets
level.Level.getBoard
board.Board.getWidth
board.Board.getHeight
board.Board.squareAt
board.Board.withinBorders
board.Square.getOccupants
com.google.common.collect.ImmutableList<E>.copyOf
level.Level.LevelObserver.levelWon
Launcher.getGame
ui.PacManUiBuilder .build
ui.PacManUiBuilder.ad dStartButton
ui.PacManUiBuilder.add StopButton
java.util.Map<K, V>.put
ui.PacManUI.Pac ManUI
javax.swing.JFframe.set DefaultCloseOperation
java.lang.IllegalArgumentExcept ion.IllegalArgumentException
java.lang.System.getSecurityManager
java.Lang.SecurityManager.checkExit
java.awt.Component.firePropertyChange
ui.PacKeyListener.P acKeyListener
java.awt.Component
ui.ButtonPanel.Button Panel
javax.swing.JButton.JButton
java.util.Map<K, V>.get
ui.Action.doAction
java.awt.Component.re questFocusWindow
java.awt.Container.add
ui.ScorePanel.ScorePanel
java.util.List<E>.site
java.awt.GridLayout.GridLayout
java.awt.Container.setLayout
javax.swing.JLabel.JLabel
java.awt.Container.add
java.util.Map<K, V>.put
game.Game.getPlayers
ui.ScorePanel.setScoreFormatter
ui.BoardPanel.BoardPanel
Level.Level.getBoard
game.Game.getLevel
board.Board.getWidth
board.Board.getHeight
java.awt.Dimension.Dimension
javax.swing.JComponent. setMinimumSize
javax.swing.JCoponent.setPreferSize
javax.swing.JFrame.getContenPane
java.awt.Container.setLayout
java.awt.BorderLayout.Border Layout
jaba.awt.Container.add
java.awt.Window.pack
ui.PacManUI..start
java.awt.Window.setVisible
java.util.concurrent.Executo rs.newSingleThreadSchedul edExecutor
java.util.concurrent.Scheduled Ececutorervice

(i).There are 6 levels in the call graph.

(ii)The six levels are makeGame, withDefaultButtons, addSinglePlayerKeys, getGame, build, start. The first one, makeGame, is to build some factories to produce ghosts, sprites and boards in every level. The second one, withDefaultButtons, is to add the ***start*** and ***stop*** buttons on the UI page. The third one, addSinglePlayer, is to establish the game rules about movements of player. The forth one, getGame, is to prepare for giving function to the ***start*** and ***stop*** buttons. While the fifth one, build, is to give function to the ***start*** and ***stop*** buttons with the preparation of addSinglePlayer, and build UI as well. The last one, start, is to start a game.

Why we decide the six levels is because when building a game JAVA application, we need first a background which includes player, boards, NPCs, and layout in every level, then design the game rules like movements and which direction is accessible. Besides, an UI is neccessay. In addition there is correspond action when player click the button, which means we need to give functions to the buttons, and when all work is done, we start the game.

(iii)

The system designs a game where the player can control the movement of the spirit by keyboard, at the same time, the player needs to avoid having contact with the NPCs, which would lead to the end of the game. And the spirite need to eat all the pellets on the panel to win the game, and the score board would show the player's score. To control the process of game, two buttons, ***start*** and ***stop*** are provided.

First of all, the system uses the pattern of factory(i.e. playerfactory, gameFactory, boardFactory and levelFactory) to make objects, which is more safe in development. To realize this game's function, the system designed several class and run the main method in class launcher. For the players and NPCs, the system designs classes like Ghosts, Player and interfaces Sprites, with a lot of functions and properties to control their behavior(i.e. Ghostcolor to decide the color of Ghost and sprites.getGhostSprite(GhostColor.CYAN) to produce a ghoast witrh certain color), and enum ***direction*** for their movements. Besides, for the game panel, the system designs the classes like panel and board, etc, with several properties to control their appearance(i.e. x, y, height, width, render() to render the board on the given graphics context to the given dimensions). As for whole UI, there is PacmanUI, which includes a score panel at the top, displaying the score of the player(s),a board panel, displaying the current level, i.e. the board and all units on it, a button panel, containing all buttons provided upon creation.

Q1

CheckMan	

1. Store states of all CheckMan (i.e., red player or black player; kings or pawns);
2. Change states of CheckMan (i.e., kings or pawns)
3. Save the positions of all CheckMan
	Play;
	Logic;
	Board

Logic	
1. Judge the validity of players 'inputs (i.e., whether to jump); 2. Determine whether CheckMan disappears or not (eaten by the opponent); 3. Judge if the locations of CheckMan are on edges of both sides of the checkerboard;	Play; CheckMan

Board	

1. Draw the checkerboard;
2. Refresh the checkerboard
	Play;
	Logic;
	CheckMan

Play	
1. Store Players' inputs; 2. Judge winning or losing (red player and black player);	Logic; CheckMan

Q1:
Rules: The checkerboard is an 8x8 grid of squares in the checkerboard pattern with black and white interlacing on it. The checkers to be used shall be and red and black in color. The pieces shall be placed on the dark squares. The starting position is with each player having twelve pieces, on the twelve dark squares closest to the player's edge of the board. When one player's pieces first all disappear, it loses, and the other player wins.

We decide to start from Play class and fill the blanks during the design process. Just like playing a video game, we need to display the game interface on the screen. Then, we plan Board class (collaboration) the responsibility to draw the checkerboard. When the first player inputs, the validity of movement is judged by class Logic (collaboration). And we should determine which player moves next. According to this game's rules, the player can give a single move, a single jump or a multiple jump move and captures an opponent's piece by jumping over it, diagonally, to an adjacent vacant dark square. So, this Logic class also decides whether the opponent's captured piece is removed from the board or not, at the same time, the Board class (collaboration) will refresh the board. In addition, when a single piece reaches the king-row, the furthest from the player, its state changes from a pawn to a king. How can we change the states of CheckMan? We need a container. Therefore, we use the class CheckMan to store states (i.e., red player or black player; kings or pawns) and positions of all CheckMan, and change states of CheckMan (i.e., pawns to kings). Finally, when one player loses his all CheckMan, he loses, the other wins. Game is over.

Q2

Our main classes of our project are class Logic and CheckMan, the former is mean to decide the game rules and the latter is mean to decide states of checker pieces: when the checker piece is valid to move, its state will be changed. What the states of pieces are, when pieces move and how pieces' states change during movements are the main parts of this game design. It's the foundation. The first responsibility of Logic is to judge the validity of players' inputs. The player cannot move anywhere they want. They can only move diagonally on the black parts within borders of the checkerboard when it is his turn. When there is a checker piece in the diagonal direction where the player wants to go, he cannot move. They can choose a simple move, a single jump move or even multiple jump move. The Logic decides that the player must take the jump if one jump is possible but can choose among several choices of jump moves if there is any. After the red player moves, the white player inputs. So one of the CheckMan's responsibility is to store the status of players (red or white): one cannot move twice continually. The class Logic also decides when checker pieces are eaten by opponents. If the pieces are eaten, they should be removed from the checkboard. This function is realized by CheckMan (collaboration), because one task of CheckMan is to save and change states of checker pieces. According to the game rules, when checker pieces arrive at the edges of the board, they will become kings and their rules will change. While Logic determines whether they turn to kings, CheckMan refreshes their states (kings or pawns), and Logic decides their new movement regulations (collaboration).

Q3

CheckMan	
1. Draw the checkerboard; 2. Store states of all CheckMan (i.e., red player or black player; kings or pawns); 3. Change states of CheckMan (i.e., kings or pawns) 4. Save the positions of all CheckMan	Play; Logic

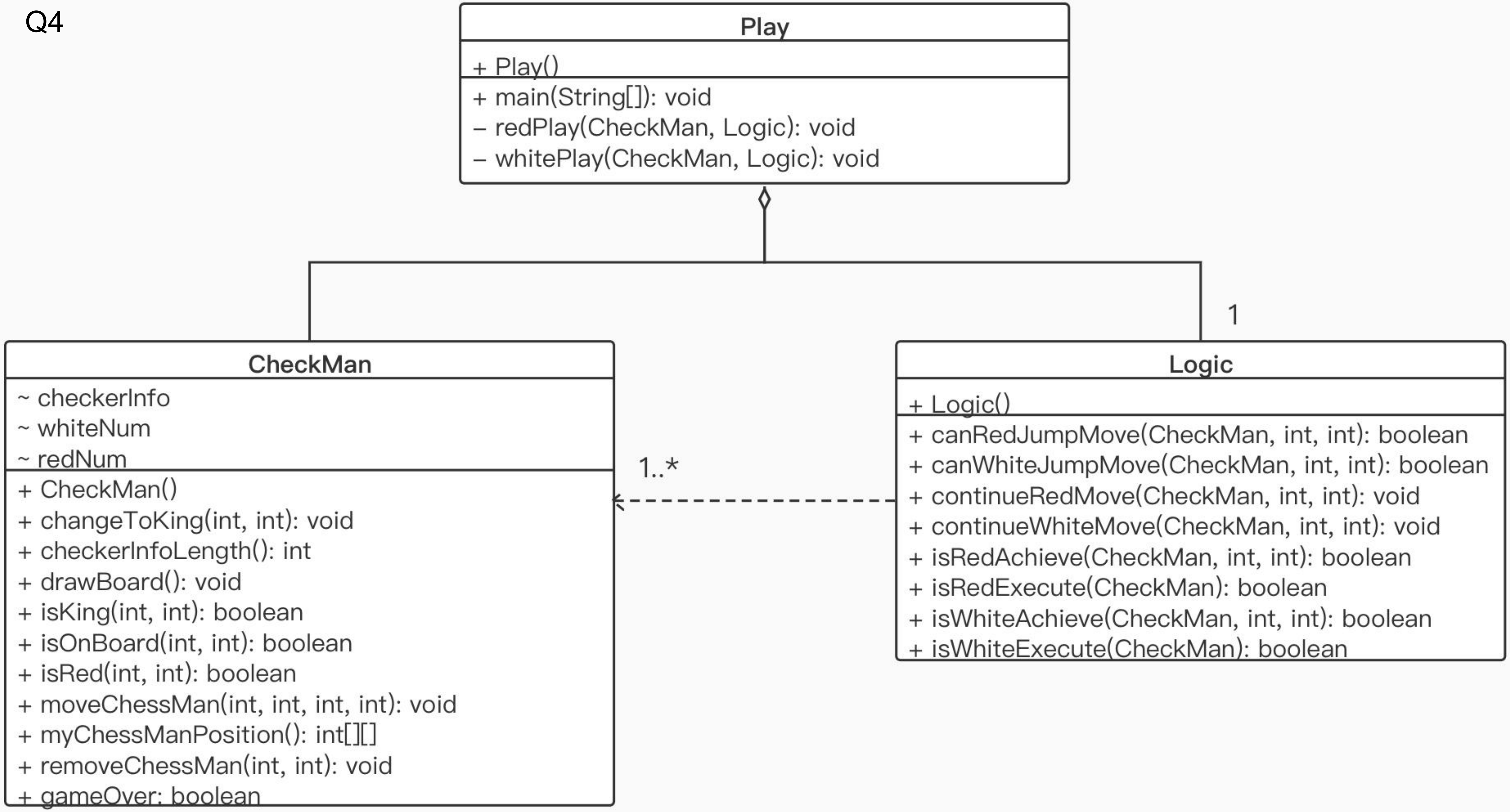
Logic	
1. Judge the validity of players 'inputs (i.e., whether to jump); 2. Determine whether CheckMan disappears or not (eaten by the opponent); 3. Judge if the locations of CheckMan are on edges of both sides of the checkerboard;	Play; CheckMan

Play	
1. Store Players' inputs; 2. Judge winning or losing (red player and black player);	Logic; CheckMan

Q3:
Board class is less important here. It can be merged with CheckMan class. From the Figure above, the new CRC cards have been shown. Now we have three main classes in our code: Play classes, Logic classes and CheckMan classes. And the process is following:

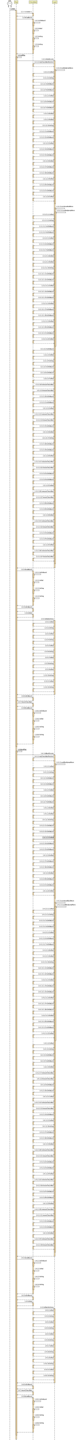
We decide to start from Play class and fill the blanks during the design process. Just like playing a video game, we need to display the game interface on the screen. Therefore, we assign CheckMan class (collaboration) the responsibility to draw the checkerboard. When the first player inputs, the validity of movement is judged by class Logic (collaboration). And we should determine which player moves next. According to this game's rules, the player can give a single move, a single jump or a multiple jump move and captures an opponent's piece by jumping over it, diagonally, to an adjacent vacant dark square. So, this Logic class also decides whether the opponent's captured piece is removed from the board or not. In addition, when a single piece reaches the king-row, the furthest from the player, its state changes from a pawn to a king. How can we change the states of CheckMan? We need a container. Therefore, we use the class CheckMan to store states (i.e., red player or black player; kings or pawns) and positions of all CheckMan, and change states of CheckMan (i.e., pawns to kings). Finally, when one player loses his all CheckMan, he loses, the other wins. Game is over.

Q4


Play
+ Play()
+ main(String[]): void
– redPlay(CheckMan, Logic): void
– whitePlay(CheckMan, Logic): void
1
CheckMan
~ checkerInfo
~ whiteNum
~ redNum
+ CheckMan()
+ changeToKing(int, int): void
+ checkerInfoLength(): int
+ drawBoard(): void
+ isKing(int, int): boolean
+ isOnBoard(int, int): boolean
+ isRed(int, int): boolean
+ moveChessMan(int, int, int, int): void
+ myChessManPosition(): int[][]
+ removeChessMan(int, int): void
+ gameOver: boolean
1..*
Logic
+ Logic()
+ canRedJumpMove(CheckMan, int, int): boolean
+ canWhiteJumpMove(CheckMan, int, int): boolean
+ continueRedMove(CheckMan, int, int): void
+ continueWhiteMove(CheckMan, int, int): void
+ isRedAchieve(CheckMan, int, int): boolean
+ isRedExecute(CheckMan): boolean
+ isWhiteAchieve(CheckMan, int, int): boolean
+ isWhiteExecute(CheckMan): boolean

Q5

there is a png file for sequence diagram in zip, if this isn’t clear enough. Thanks a lot!



Q5：The main elements for sequence diagram is the judgements of chess movement and execute of the chess, they are asynchrony and constraints. Only when the system judges that, the CheckMan can move, the chess will execute.

In sequence diagram, when the players open the game, the board is drawn. Then, red Checkman plays first, and the system will check whether the move is legal. If it is, red Checkman moves or jumps. The the board refreshes. Then white Checkman plays, and the system judges whether it is legal. After a series of judgments and movements, the game will give the result based on the rules we mentioned before .